1

File Processing Overview

CHAPTER OBJECTIVES

When you complete this chapter, you will be able to:

- Understand the user's perspective on file processing and why file processing is critical to overall system performance
- Discuss the impact of file processing on the performance of both user and system programs
- Understand the standard file terminology
- Understand and be able to discusss the key issues in file design
- Identify the key I/O performance criteria

1.1 OVERVIEW

When most programs are written, they usually include statements such as READ or WRITE. Although languages vary on the specific syntax, the concept is the same. Specifically, the user is trying to READ data from, or WRITE data to, some kind of medium for permanent storage. What *really* happens next? How does the "system" know what to do? How does the user's program know when the READ or WRITE request has been completed so that the program can resume processing?

These, and other questions, are all related to input/output (I/O) which deals with data transfers in to, or out of, the main memory of the computer. The term

"I/O" is analogous to the terminology of READ/WRITE. As can be seen in Figure 1-0, *The I/O interface between the user's program and the peripheral devices is known as the file system.* I/O will be analyzed in detail, both from the user's point of view and from the file system's perspective, with the following overall goals:

- Understand how each of the hardware and software I/O components work together
- Understand the *internal* data structures required to process I/O requests
- Understand the design of the disk and tape file structures
- Understand how the data and control information flows through the entire system

We will discuss in detail what it takes to design and implement a file system. In addition, we will always keep in mind the issue of maintaining a competitive level of overall performance, or system responsiveness.

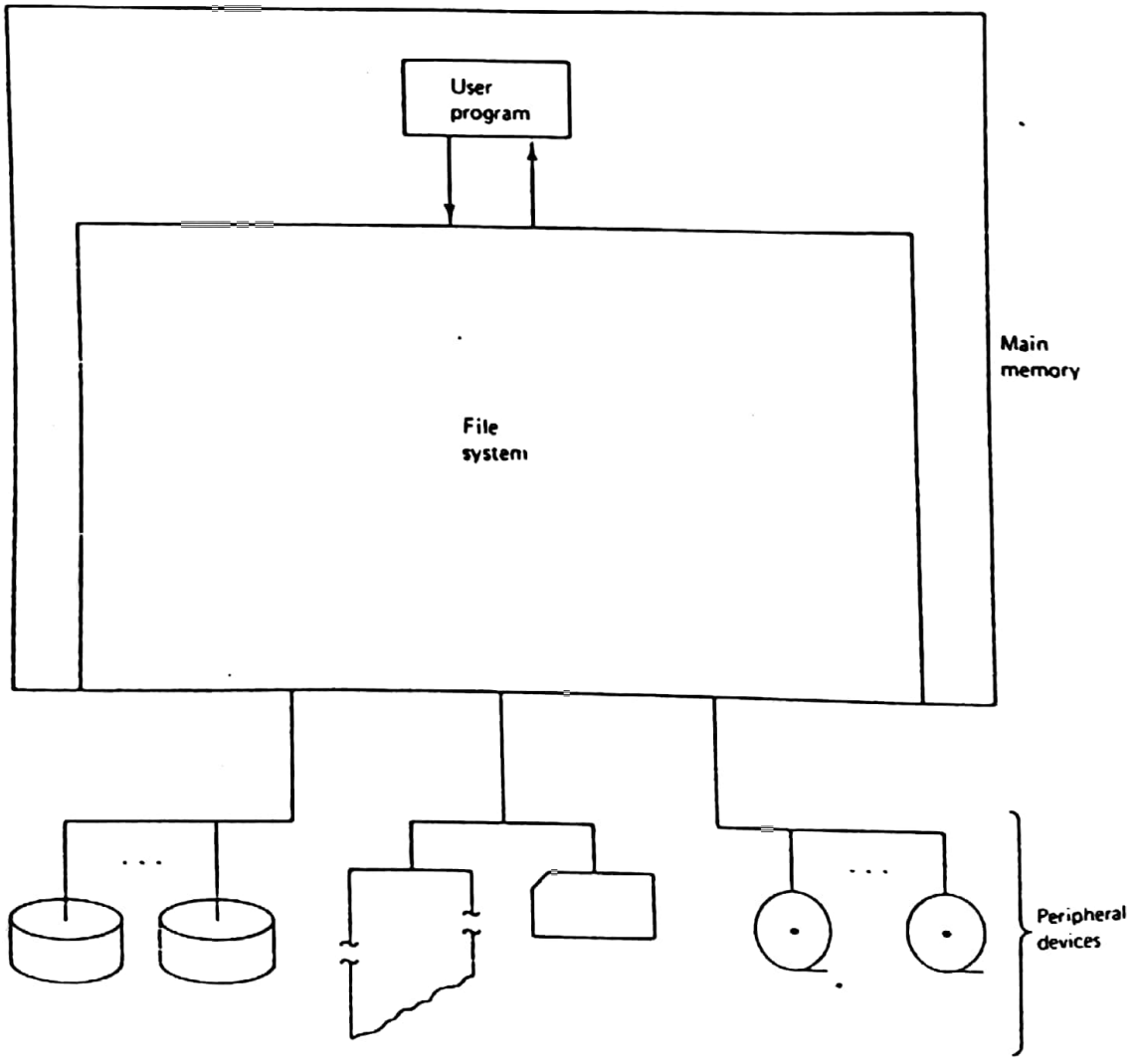


Figure 1-0 Overview of I/O.

What is most important here is to understand the *concepts and ideas* presented. This is because the industry in general, and file systems, I/O, and devices in particular, are changing so rapidly.

Finally, we must consider the growing impact of the personal computer on all activities in this industry. For example, a user may decide to run applications, such as payroll or inventory, on a large IBM mainframe computer. This machine could cost several millions of dollars and have gigabytes of on-line storage capacity. At the other extreme, a user may decide to run a payroll application on a personal computer costing several thousand dollars and perhaps having several megabytes of on-line storage. For which user do we design file systems? Can we design a system that can be beneficial to both large and small computer users? For these reasons, it is best if we can understand the whys and hows of file systems. Then we can try and apply those ideas to the current task.

1.2 WHY STUDY I/O?

The fundamental reason for the importance of studying I/O is that *I/O is slow* in comparison to the internal speeds of computers (see Figure 1-1). In fact, relative to the speed of an I/O device, the speed of a central processing unit (CPU) is increasing more rapidly. The primary reason for this is that I/O devices are mechanical and are thus significantly slower than the electronic speeds of semiconductors and microprocessors.

Second, a user may think that a single READ or a WRITE statement in a program causes only one I/O operation to occur. In reality, *zero to many* actual I/O operations could occur. This is analogous to the situation in which one line of code written in a high-level language such as COBOL can result in zero to many actual machine instructions.

There are several other reasons why I/O is important to study:

- There is much I/O occurring in a system on behalf of each user, over and above the I/O that the user specifically requested (e.g., compiling programs, editing memos, listing directories, etc.).
- The capacity of the media (e.g., tapes and disks) is increasing at about 30% per year. This means that more and more I/O can be performed on the same devices, thus causing another bottleneck at the device level.
- I/O capacity is severely limited. For example, the average time that a disk takes to mechanically reposition itself to another place on the disk is about 20 milliseconds (ms). Also, assume that we are working on a 1-mips (million instructions per second) machine. In the time it would take to process *one* I/O request, the CPU could have executed over 20,000 instructions. This is greatly exacerbated if the CPU runs at 100 mips. Then, in the span of a single I/O instruction, over *2 million* CPU instructions could have been executed!

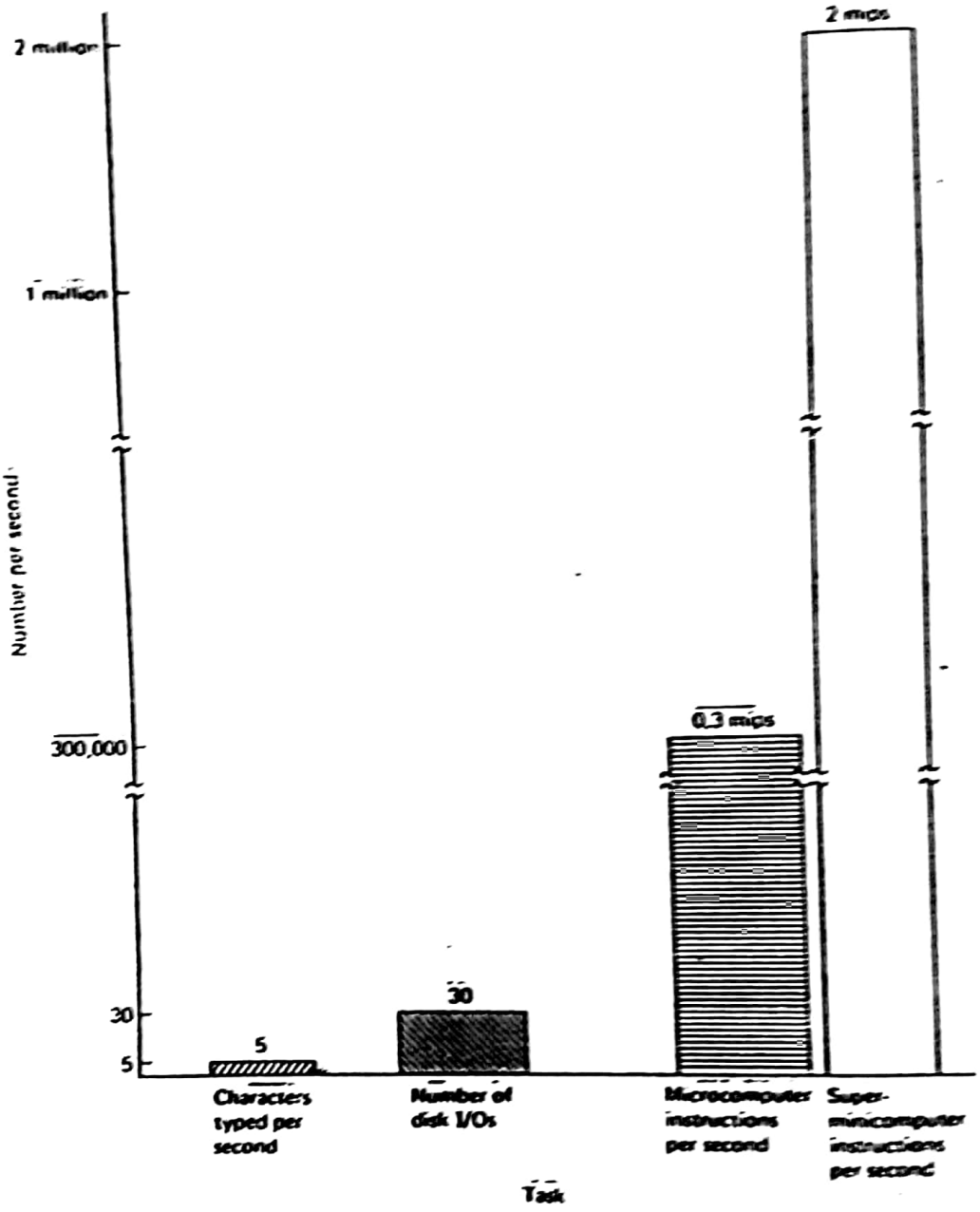


Figure 1-1 Work accomplished per second.

One reason that multiuser operating systems came into being was to try to use these CPU cycles that would otherwise go unused while a program is waiting for an I/O operation to complete. The way this is done is to allow other programs to execute while the I/O is going on.

If we perform the calculation above in a different manner, we can see another major reason that I/O is so critical. If the average I/O time is 20 ms, we can calculate that there can be at most 50 I/O operations completed per second. Due to a variety of factors, such as having multiple jobs running in the system at any give time, the maximum I/O load is rarely achieved. Usually, it is considered good if a system reaches

75% to 80% of I/O capacity at any given time. Note that this percentage includes *all* the I/O going on in the system against that specific device (e.g., paging, program loading, compiling, and other utilities). How can we possibly have 50 or more students working on the same machine? How can there be 5000 airline travel agents making reservations without running into each other?

What we have just discusssed is one of the critical constraints on the performance of any system. Specifically, the amount of I/O that can be performed in any given interval of time is severely limited. Thus *the file system must treat I/O as a precious commodity that is utilized only when it is absolutely necessary.* We shall examine this topic, and determine where the constraints really are and what can be done to solve the I/O performance and throughput problems.

There is one other observation that can be made. For reasons of greater productivity, the software development industry is moving toward the use of higher-level languages, such as COBOL, FORTRAN, or Pascal. However, high-level languages tend to hide completely from the user the amount of I/O that is really being performed. The benefit of this is that users do not need to understand in a great amount of detail how the system works in order to get their jobs done. However, it also means that users have become unaware of how to take advantage of the system and thus improve its overall performance and throughput levels.

Finally, each high-level language provides the user with a different set of I/O capabilities. Thus the functionality that a user requires will tend to dictate in which language the user will choose to write the application. Either this will happen, or the user will try to make the application work within the constraints of the language primarily utilized at the user's installation.

1.3 I/O FROM A USER'S PERSPECTIVE

What does a user expect, or require, when an I/O request is made? What does a user think will happen, and what actually will happen?

When a user runs a program, or works interactively at a computer terminal, the objective is to complete the task at hand as soon as possible. Often, *the limiting factor in the responsiveness of the executing program is the underlying I/O performance of the system.* To understand why this is the case, we need only investigate the kinds of work the user performs.

Typically, the user wants to be able to perform the following kinds of activities without having to be concerned with all the technical details:

- To write data into a file today and be guaranteed that the data will be readable tomorrow
- To create files from one higher-level language, such as COBOL, that will be readable when accessed from any other language (e.g., FORTAN, Pascal)
- To be able to access the records in the file either sequentially or randomly, as required by the job being processed

- To have a reasonable combination of functionality, performance (i.e., response time), and stability, meaning that the system is reliable and will not adversely affect the integrity of the data
- To have the flexibility of choosing when to define file characteristics and when to let the system choose the parameters itself
- To *not* have to know all the technical details of the machine to get the job done
- To have the system make optimal use of the available resources so that the user is not forced into purchasing more hardware than is required to do the job
- To have the assurance that the data will not be inadvertently "lost"

Fundamentally, the user wants to be able to complete the work that needs to be done and not have the "computer" get in the way. Unfortunately, for most users, I/O is the "black box" shown in Figure 1-0. This is because the usual contact with I/O is either indirect, through a higher-level language, or invisible, meaning I/O that is performed on behalf of the user. For example, there is the paging that goes on in a virtual memory machine; the I/O performed by the operating system to load the user's program, and the I/O to compile the user's program. It is the intent of this book to break down this mysterious "I/O black box." We will do this by examining how the various system components work together to get the I/O completed successfully, at the right time, and in the most efficient manner.

1.4 I/O FROM A SYSTEM PERSPECTIVE

The objective of any file system is to meet the data management needs and requirements of its users. If any file system does not meet these needs, users will go elsewhere in search of software that allows them to do their job properly. Thus the file system is providing an I/O service to its user base. The basic goals and objectives of a file system are as follows:

- To meet the data management needs and requirements of the user
- To guarantee to the highest possible degree that only *valid data* are ever actually written into a file
- To maximize overall system performance and throughput by minimizing the number of *actual* I/O operations that need to be performed
- To provide I/O support for multiple users, not just a single user
- To provide I/O support for a variety of different device types
- To minimize, or eliminate, the potential for lost or destroyed data (i.e., *never lose data*)
- To optimize the control and utilization of the devices installed in the computer configuration
- To provide a standardized set of I/O interface routines that either the user can call directly, or the various system modules, acting on the behalf of the user (e.g., compilers, linkers, system utilities, etc., can call

In summary, the file system must provide an efficient and effective I/O service that provides the functionality and performance required by the user and by the pressure of competition. However, no statements made regarding file processing can be "guaranteed" to be correct in every application. The best that can be done is to make us all aware of how a file system works. Then each of us can decide what is the best technique to use in a given situation. Also, it can be demonstrated that if the user has some knowledge of how I/O is performed, the performance of the user's task can be dramatically improved *without changing one line of code in the user's application.*

Not all the features and functions that are discussed in this book are accessible from higher-level languages. However, these features are available on most systems in either the monitor, file system, utility programs, job control language, or interactive system commands. As the various components of a file system are discussed, we will also explain where the required information comes from.

1.5 TERMINOLOGY

To understand what I/O and file systems are all about, it is necessary to have common definitions on which to base the discussions. One problem in the computer industry is that there is no commonly accepted and used standard set of definitions. Most software development companies use similar words, but words that have different meanings. In this book we attempt to use the most common definitions for the terminology, but you still need to be aware that different vendors and writers may use different terms or definitions.

1.5.1 Fields

A field is an elementary data item such as a name, employee number, city, and so on (see Figure 1-2). It is characterized by its size, or length, and its data type (e.g., ASCII string, packed decimal, integer, etc.). Fields can even vary in length. In this book fields are assumed to be of fixed length, since in most standard applications, as well as in most higher-level languages, variable-length fields are not supported.

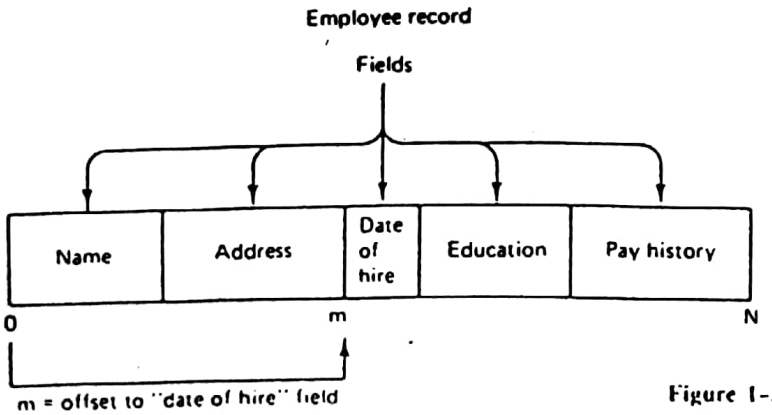


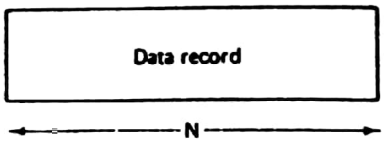
Figure 1-2 User data record structure

1.5.2 Records

A record is a collection of related fields that can be treated as a unit by some processing program. For example, you could have a person's payroll record, a student record, or a health record. All the data in the record are related to, or associated with, the same person or entry. A record can contain many fields of information, as shown in Figure 1-2. For example, within a person's employment record, one could have individual fields containing home address, telephone number, past work history, pay, and even school and degree held. How can a specific field within a record be located? *The field offset is the number of bytes from the beginning of the record to the start of the field in question.* Thus, if we know where the record begins, we can add the field offset to obtain the location of the desired field.

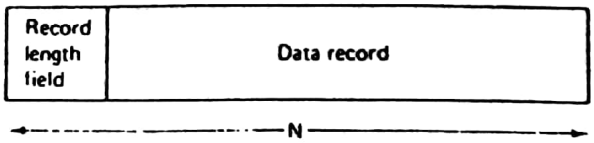
Records also have certain characteristics, such as:

- Records must somehow be created.
- Records should have a maximum defined size, although in some circumstances, not all data need be present. The reason for this is that the file system must have guidelines as to what record sizes should be considered valid and which records should be rejected. Remember, the file system must guarantee that only valid records are placed into a file.
- Records within a particular file have the same fields, field offsets from the start of the record, and field data types. This fact not only makes it easier to program, but also improves performance since the record does not have to be scanned to locate the field to be used.
- Records have a particular format. *There is one and only one record format for any specific file,* although different files can have different record formats. *If all the records within a file are of identical size, we have fixed-length records* (FLRs), as shown in Figure 1-3a.



N = size of fixed-length records

(a) Fixed-length record



N = length of entire record, including attached length field

(b) Variable-length record

Figure 1-3 Record formats.

Alternatively, *if the size of each record in a file can change, we have variable-length records* (VLRs), as shown in Figure 1-3b. VLR records can vary in size only up to some predefined maximum length. For VLR records, there must be some way in which the length of the record can be known so that the data can be written or read correctly. This is done by *appending a length field to the front of each record* in the file. Thus VLR records only take up as much room as they really require.

In contrast with VLR records, FLR records *always* take up the maximum space, whether or not all of the fields have data in them. Finally, fixed-length records do *not* need a length field, because all records are the same size.

1.5.3 Files

A file is a collection of related records. For example, in a payroll file, each employee has one data record, and all the fields in that record relate to that employee. Similarly, all of the records in the file are related since they all contain payroll data. I/O is concerned with the reading and writing of data records between a user's program and the peripheral device. All related records are stored in a structure called a *file* (see Figure 1-4). This is analogous to a file cabinet. Here, someone puts data (i.e., file folders) into the cabinet (i.e., file) in an organized way so that the information can later be retrieved (i.e., read), updated, and put back into the file (i.e., write).

A file in a computer system is a structure designed to contain data (e.g., file folders). The data are organized in such a way that they can be easily retrieved, updated or deleted, and put back into the file with all changes made. Just as we can set up many different filing systems, there are many ways in which we can organize the data in computer-based files so that they can be used by applications and other programs, as well as interactively from a terminal.

There are several characteristics of files that are important to understanding them. Files can be *created* or *deleted* by the user. For example, you might create a personnel file, an inventory file, or a work file that you will use for a short period and then delete because that file is no longer needed.

Files have unique *filenames.* When a particular file is wanted, it is only necessary to give its name and the file system will locate it. However, filenames must be unique, or the file system will not know which is the correct version of the file wanted by the user. *The entire unique filename is known as the file's pathname.* The pathname consists of the following information:

$$\text{pathname} = \text{device} + \text{user} + \text{directory} + \text{filename}$$

The *device* parameter is required since there could be many different devices connected to the system at any one time. Therefore, on which device does the file reside? Typically, the device need not always be specified since the default can be the device on which the major directory, or account, resides.

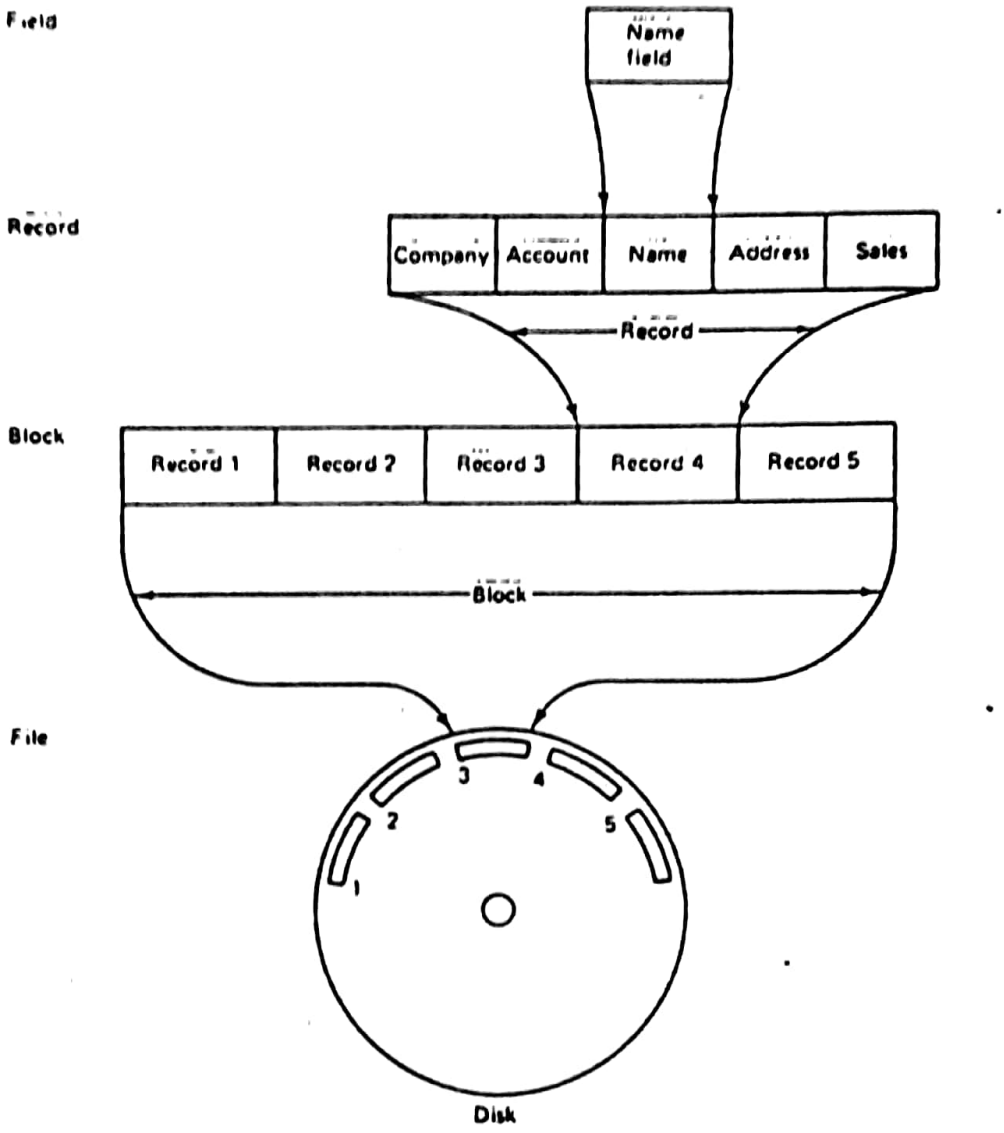


Figure 1-4 Fields, records, and files.

The *user and directory* parameters are needed to adequately define the location of the file on the device. For example, there could be several files named "ABC." However, each one belongs to a different user and/or resides in different directories. The combination of device, user, and directory specifies the unique directory in which the file called *filename* exists (see Figure 1-5).

Just as we can have an inventory file, or a personnel file, the *filename* defines not only which file the user wants out of all the files that may exist, but also implies a *location* and *size*. Of all the file cabinets we may have, when someone wants the "parts file," we know exactly where to look for it. In a similar manner, the file system maintains within a directory entry the file name, its location on the medium, and the size of the file itself (i.e., the number of blocks written into the file).

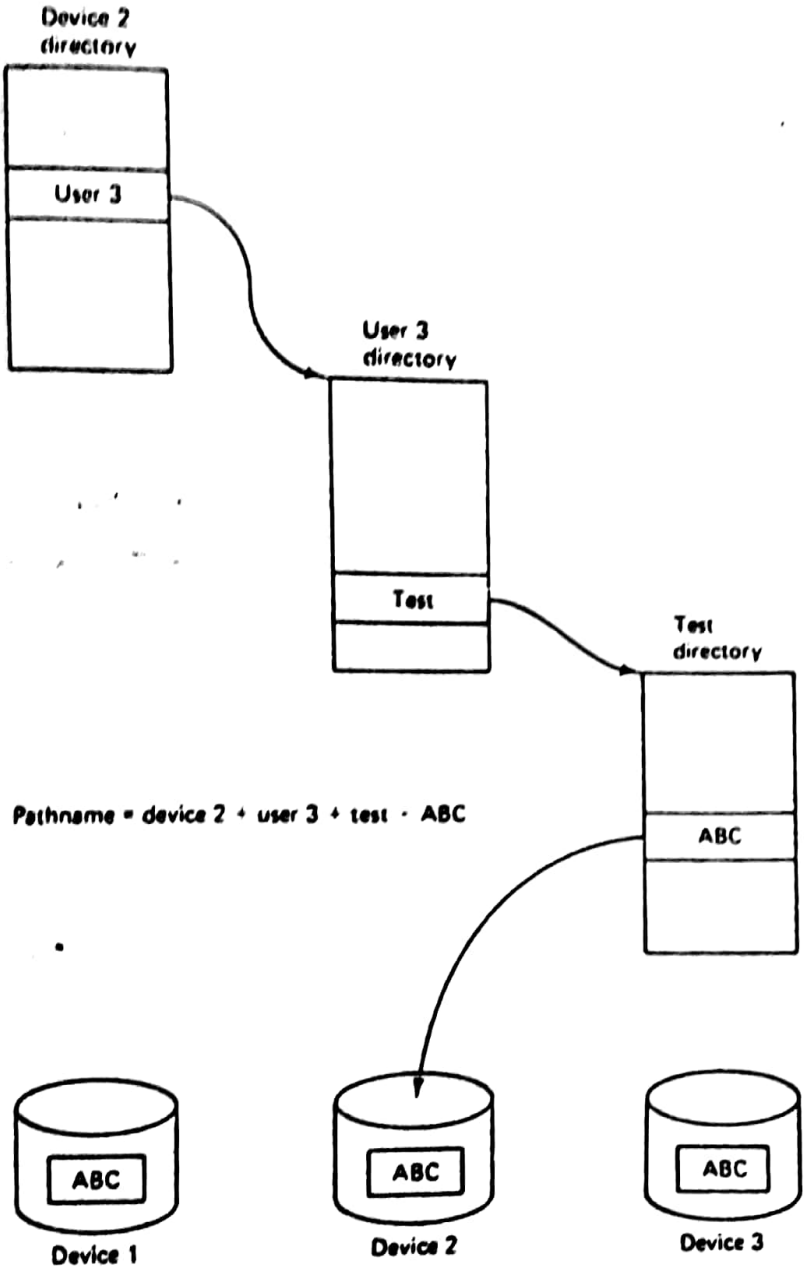


Figure 1-5 Pathnames to files.

Files can also be *shared* or *nonshared* between multiple users. Some files may contain sensitive data, and any access to the data should be restricted (e.g., a personnel system's files). Other files, such as those files used in an airline reservation system, need to be shared by all the travel agents and airline ticket offices around the world. Sharing can mean that only certain users will be allowed to READ or WRITE data to the actual file, or that no user has access to the data until the owner allows access.

The last file characteristic that will be discussed is that of performance. *Performance is the user's perception of the system's responsiveness.* Also, it is seen as how long the software can work until it breaks. Depending on how you organize the data within a file, you can retrieve data, or insert new data, slowly or quickly. Sometimes we may want to look at all records in the file so as to organize the records in such a manner that it is a fast process to sequentially access the next record. On the other

hand, the records in a doctor's office need to be rapidly accessed by patient's last name in a completely *random* manner. We will learn that once it is known how the data in the file will be used, we can then design a means of organizing those records to get the performance and access capabilities that are required by the user of those data.

1.5.4 Blocks

A block is the unit of data that actually gets transferred (i.e., READ or WRITTEN) to, or from, the user's file that is located on a particular device. A block is the smallest amount of data that can be transferred with a single I/O operation. A block may contain one, or more, records. In fact, a block may contain only a partial record, so that to access the entire record, more than one block will need to be read.

This raises a question as to whether a block can vary in length. If this were the case, we could force all records to fit within a block simply by increasing the size of that block. As was the case for records, blocks can be either fixed or variable in length. If the block is variable in length, a *block length field* must be appendded to the front of each block. Also, to be able to read and write entire blocks into main memory, we will require that blocks be allowed to vary only up to a predefined *maximum block size*. The file system will then be able to allocate enough space in main memory for I/O at least at large as the maximum block size. For fixed-length blocks, the maximum block size is equivalent to the specified fixed length.

Figure 1-6 shows two different conditions that can occur when putting records into a block. In some file structures, records are not required to be smaller than the maximum size of a block, and in fact could be of almost unlimited length. When records are allowed to cross block boundaries, we say that the "system" allows *spanned* records (see Figure 1-6b and d).

In this situation, all blocks then become of maximum size. Any records that do not completely fit within the block are divided between two or more blocks. Thus, with spanned records, there is no wasted space within a particular block. However, as we shall see later, the price for not wasting space in the file is paid with slower access times. This is because a *single I/O* operation (e.g., read or write a block) may not be enough to transfer the entire record. Thus at least one more I/O operation will be required.

Alternatively, where records are prohibited from crossing, or spanning, block boundaries, we say that the "system" supports *unspanned* records only (see Figure 1-6a and c). In this case, the maximum record size must fit completely within a block. In addition, the file system will pack as many records in each block as can possibly fit. If the last record in the block can fit only partially, it must be completely moved into the next block. In this situation, file space is potentially wasted, since not every byte of the block may be used (see Figure 1-6a).

The goal of a file system is to be able to READ and WRITE records quickly. One way to accomplish this is always to guarantee that when a block is READ, it contains one or more *complete* records (i.e., do not allow spanned records). Thus

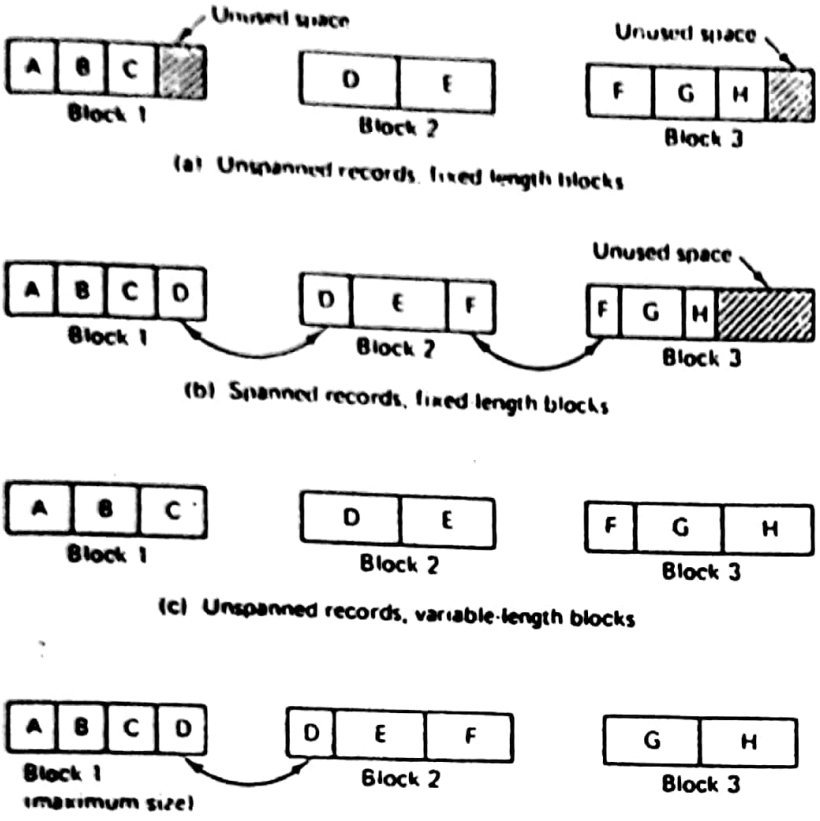


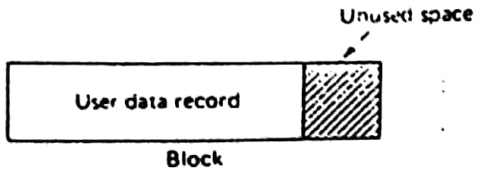
Figure 1–6 Spanned and unspanned records.

multiple I/O requests will not required to READ or WRITE any record in the file, and time will be saved.

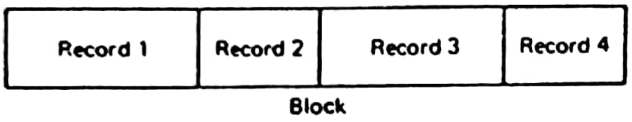
On the other hand, it can be argued that by allowing records to span blocks, you can package the records together more densely and without wasted space (see Figure 1–6b and d). Unspanned record structures always run the risk that space will be wasted because there could be room in the block, but not enough room to hold the next record. Thus space is being wasted. Here again, we are trading off optimal use of media space against reducing the number of I/O operations required to READ or WRITE any record in the file. In other words, we are trading media space for improved performance and user response time.

Another characteristic that is important in relation to file system performance is the concept of a *blocking factor.* For example, if a file had fixed-length records, and if the block size were known, we could calculate how many records could fit into any block in the file. *The number of records that can fit into a block is called the blocking factor* (see Figure 1–7). If we had variable-length records, or even spanned records, we could not predict with 100% accuracy what the blocking factor for that file would be. However, we should be able to determine how many records *on the average* will fit into any given block in the file, and therefore calculate the blocking factor for the file.

The importance of a blocking factor, or more precisely, the placing of more than a single record into a block, may not be fully understood at this point. However, to illustrate, let us assume that we had two files. One of them wrote only one record



(a) One record per block (blocking factor = 1)



(b) Multiple (N) records per block (blocking factor = N)

Figure 1-7 Blocking factors.

into each block (i.e., a blocking factor of 1). The other file had a blocking factor of 10 (i.e., there were 10 records per block). If each file contained 1 million records, the second file would require 900,000 *fewer* I/O operations to read all the records! It is not uncommon on microcomputers to have disks with an average access time of 90 milliseconds. If this were the case, the first file would take about *24 hours longer* to read all the records in the file.

A significant point to remember is that a reasonable blocking factor greater than 1 will always improve overall performance and system throughput, by cutting down on the amount of I/O that the system must perform. However, we need to be sensitive to the impact of block size on main memory. Memory allocated to data blocks is unavailable for program use. Thus we need to make a trade-off between improved performance gained by using larger block sizes and the amount of main memory available for program use.

1.5.5 File Attributes

How can we determine the size of the records in a file containing only fixed-length records, since the records do not have a length field? Or, to put it another way, how can the file system know what the size and format of the records are in the file that was created last week? *There are certain characteristics that define precisely what a file is and how the records in that file must be processed. These characteristics are known as the file's attributes.* Specifically, some file attributes that we have all ready discussed are the following:

- Filename
- Maximum record size
- Record format (i.e., VLR or FLR records)
- Maximum block size
- Block format (i.e., fixed- or variable-length format)
- File size
- File location

Since a file's attributes are critical, they are permanently recorded within the file. Some of the common attributes, such as filename, may be stored within the directory entry. Then the *file specific* attributes (e.g., key size) can be stored within the file itself.

1.6 KEY ISSUES IN FILE DESIGN

As we examine various techniques for organizing user data records within a file structure, we must always keep in focus our overall goals and objectives. It makes no sense at all to design a file structure, and to write all the supporting code, if the resulting file is not what the user needs. When this situation occurs, as it does far too frequently, the user will simply not use the product—no matter how sophisticated and elegant the design. To maximize the probability of designing a file structure that meets the user's needs, we need to keep in mind the following overall guidelines, which are listed *in order of decreasing priority:*

1. Meet the needs and requirements of the *user*
2. Minimize the number of *actual* I/O operations performed
3. Trade-off of functionality versus performance versus resources
4. Never lose data
5. Use the 80/20 rule when deciding costs versus benefits of some feature

In making the trade-offs noted above, we must understand that unlike some other disciplines, there are no right answers to such decision guidelines. The only "cookbook" approach that can be used is to keep focused on these issues.

1.6.1 Needs and Requirements of the User

This point is so obvious that perhaps it should not even be mentioned. For example, let us assume that we went out to purchase a brand new automobile. For most of us, the cost of such a purchase is quite significant relative to most of the other kinds of purchases we make. Therefore, we carefully set out to find a car that meets our requirements. We may not be formal about this process and actually write down a list of requirements, but we have at least the major ones tucked away in the recesses of our mind. By a comparison process, we drop dealer after dealer from our list of possibilities. We may do this because of styling considerations, reputation of dealer, availability of the car, or other factors. Nevertheless, what we have done is to not even consider a broad range of products for one reason or other.

In the same manner, a computer user will shop for software or hardware and will decide *not* to purchase a product for any number of reasons. High on that list, however, is the perception that the product does not meet the needs and requirements of the particular user. Also high on the list are those products which have, correctly or incorrectly, a poor reputation for performance, quality, functionality, or respon-

siveness to customer requests. In summary, if the user's requirements are not met, nothing else matters, since the product developed will not be purchased.

1.6.2 Number of Actual I/O Operations

This is a critical issue, since the number of *actual* I/O operations needed will directly affect the overall performance as seen by the user. An actual I/O operation means one in which we have to go to the device itself to transfer the data. We will discuss this more later, but many I/O requests are processed between buffers in main memory and never get out to the device itself. For now, it is sufficient to know that devices are mechanical in operation. Thus it takes a comparatively long time to access a device compared to the time it takes to execute a single CPU instruction.

In addition, consider that one line of code written in a higher-level language, such as FORTRAN, COBOL, and Pascal, will actually generate one to many actual CPU instructions. In a similar vein, a simple READ or WRITE from the user can result in zero to many actual I/O operations. Thus designers of file systems need to be aware that the actual number of I/O requests should be kept at a minimum to provide the user with the "best" level of performance.

1.6.3 Functionality versus Performance versus Resources

Whenever we design and implement any project, we always end up consciously, or even unconsciously, making design and implementation trade-offs. Maybe we are running behind schedule, so we need to drop some functionality, not achieve all our performance goals, and so on. In any case, we must be acutely aware of what can and cannot be traded off.

First (see Figure 1-8, step 1), we must provide the user with at least the amount of functionality that the user requires, and possibly more if we are to be fully competitive against the other file systems on the market. Functionality can mean many things to different people. Here, functionality will mean not only the features provided to the user via some programming language, but also the quality of the error messages that are generated and how "user friendly" we can make our system. We should never trade off so much functionality that our product becomes non-competitive. Thus we must walk the fine line between developing a competitive file system, while not making it overly complex so that it is never completed.

Performance is generally second in line behind functionality in importance to the user (see Figure 1-8, step 2). Performance will not become an issue until the user is satisfied with the amount of functionality that has been provided. Once this level of functionality has been attained, the user will then demand that the system "go faster"!

In a file system, this generally means that we must cut down the amount of time that it takes before we can give the user a response to an I/O request. This time interval is directly related to how well we can minimize the actual number of I/O operations that are performed. There are buffering and blocking techniques, as well as several other techniques, that can be used to reduce dramatically the amount of

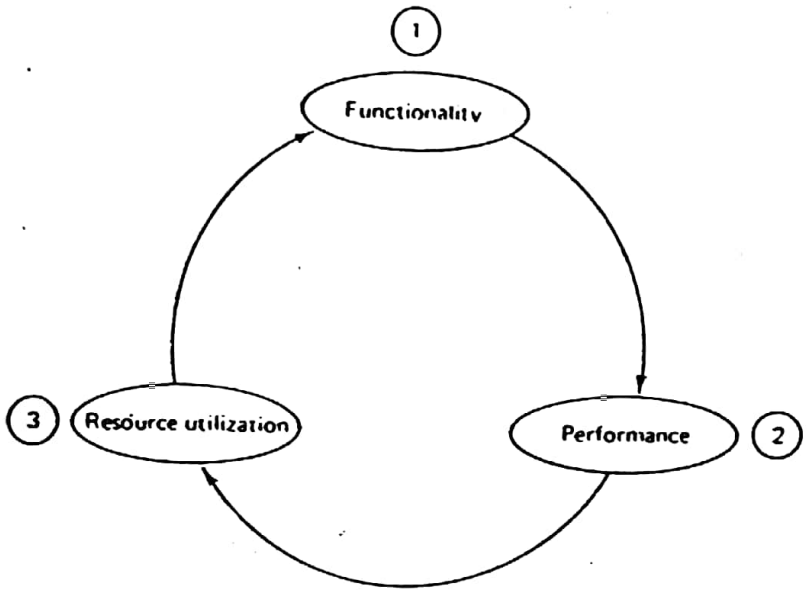


Figure 1-8 File system trade-offs.

time we spend performing I/O. Usually, we must be concerned with how fast we can READ a record for the user, or conversely WRITE a record for the user. If we can do this well, the user will tend to be more satisfied.

Finally, if our file system meets the functionality and performance expectations of the user, we must be concerned about how much of the user's resources are required to use our file system properly (see Figure 1-8, step 3). Resources that must be analyzed are as follows (see Figure 1-9):

- The amount of *main memory* (a) needed for each user
- The *code size* (b) of the file system and operating system, which can affect how much main memory is left for the user's programs
- How well we use the *media space* (c) on the peripheral devices (e.g., disk space)

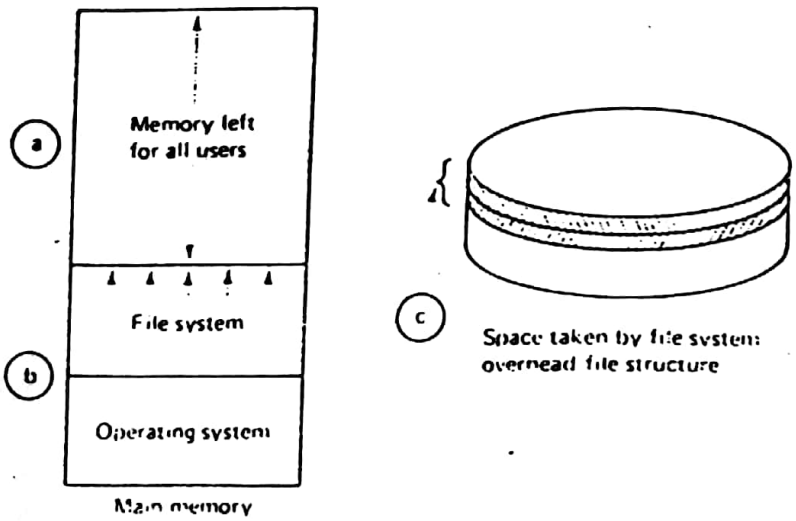


Figure 1-9 File system resources.

Resources cost money. In particular, mechanical devices such as disks and tapes cost relatively large amounts of money, and the file space available tends to be limited. Available media resources form a hard constraint on what we can, and cannot, assume exists at a user's installation. Therefore, the file system must use the space and resources as best it can.

In summary, the user's highest priority is functionality, followed in order by performance and then resources. The inherent difficulty in this task is that we must achieve the functionality and performance levels required by the user *within the constraints of the resources provided.*

1.6.4 Loss of Data

The reason that users want to WRITE their data onto various media, such as disks and tape, is so that they can access the data whenever they need them. They also know that by saving the data in a "computer" they will be able to access any piece of data significantly faster than if the same data were kept in traditional filing cabinets. However, if they ever discovered that a file system, no matter how sophisticated, occasionally and mysteriously lost data, they would drop it like a hot potato.

Consider for a moment what would happen if a file system lost only one piece of data in a file of a million records. The system is 99.999% accurate! No user would complain if we could guarantee this level of accuracy, would they? What would happen if that lonely piece of lost data was *your* payroll check?

1.6.5 The 80/20 Rule for Deciding Costs versus Benefits

It is always a difficult problem to try to decide what to do next when making a trade-off, either one such as that described in the preceding paragraph, or any other. A nineteenth-century Italian economist named Pareto developed a principle that is just as applicable today as it was then. What he hypothesized can be referred to as *the 80/20 rule.* What this states is that *for a wide range of activities, 80% of the results are generated by 20% of the tasks.* For example, 80% of a company's revenues are generated by 20% of the salespeople. Eighty percent of the code written in a software organization is written by 20% of the programmers. Or to be more specific to file systems, 20% of the functionality will be used 80% of the time. Twenty percent of the code written has 80% of the performance problems, and so on. Although the percentages may vary somewhat, they tend to be consistent over a broad range of activities.

What is important here is not how precise the rule is, but what it is really trying to say. From the perspective of file system design, it is important to do an analysis to find and identify that critical 20% of the functionality which most users want; to locate that 20% of the code that has most of the performance problems; and to test the reliability of our file system to find the 20% of the modules that will contain the vast majority of the bugs.

These questions and issues can be answered by performance analysis tools, histograms, test suites, and a careful analysis of customer needs over both the short

and the long term. That result of spending the effort to get this information will be that the file system developed will be efficient, stable, and competitive. The bottom line is that designing any software product in general, but more specifically a file system, encompasses much more than the code itself.

1.7 KEY FILE DESIGN RATIOS

As we proceed through our analysis of various file organizations, it will frequently be necessary for us to make a series of trade-offs to accomplish our tasks. Although there are no "right answers" that are guaranteed to be correct in all situations, we do have for analysis purposes two key formulas that help us to determine what kinds of trade-off decisions to make. These two ratios, however, only provide some insights into how records are expected to be processed within a file [3,4]. Thus they are really only inputs into our file design decisions.

1.7.1 File Activity Ratio

The file activity ratio, also known as the hit ratio, provides us with a measure of how many records in a given file are typically accessed in any job. Specifically, the file activity ratio is defined as follows:

$$\text{file activity ratio} = \frac{\text{number of records accessed}}{\text{number of records in file}}$$

If, in any given job, the ratio approaches a value of 1, it indicates that almost every record in the file is READ by the job. Thus sequential performance, or the ability to READ all records in a file, becomes critical to overall performance.

On the other hand, when few of the records are accessed in any given job, the ratio approaches zero. When the activity ratio is small, it means that most of the records in the file are *not* READ, only very specific ones. Thus *random* access to data records becomes a critical attribute toward high performance.

Finally, if the ratio is somewhere in the middle, we must look for other measures of performance and functionality with which we can make our design decisions. What is important, however, is that we understand how a file will be used.

1.7.2 File Volatility Ratio

The file volatility ratio is a measure of how fast the original version of a file is changing. If the file is changing rapidly, we should aim to make our change routines (i.e., WRITE and DELETE) as fast as possible. On the other hand, if the file is not changing rapidly, READ performance can be emphasized.

Specifically, the file volatility ratio can be defined as follows:

$$\text{file volatility ratio} = \frac{\text{no of additions} + \text{number of deletions}}{\text{number of records in original file}}$$

If the file has many records that have been added or deleted since the original file was built, this ratio will be high (i.e., approaches 1). This indicates that the *change functions*, WRITE and DELETE, must be designed and implemented for high performance.

On the other hand, a low volatility ratio (i.e., one approaching zero) means that overall the file is not changing very rapidly. Thus the *nonchange functions*, such as READ, should have a high performance. Finally, a ratio somewhere in the middle should force users to look for other measures to make the correct design and implementation decisions.

1.8 SUMMARY

The purpose of this chapter has been to introduce the topic of I/O. We examined I/O not only from the vantage point of the file system, but also from the perspective of the user. It is critical to understand both of these viewpoints to design a file system that meets the needs of both parties.

Finally, we discussed some of the important definitions that will be used throughout this book. These definitions, as well as some file system analysis ratios, will become the basis for our future discussions on the design of file systems.

KEY WORDS

Block
Blocking factor
80/20 Rule
Field
Field offset
File
File activity ratio
File attributes
File system
File volatility ratio
Fixed-length records (FLR)
Input/output (I/O)
Maximum record size (MRS)
Nonshared files
Pathname
Performance
Record
Record formats (RFM)
Shared files
Spanned records
Unspanned records
Variable-length records (VLR)

SUPPLEMENTAL REFERENCES

1. Claybrook, Billy G., *File Management Techniques*. New York: John Wiley & Sons, Inc., 1983, Chap. 1. This book provides a general introduction to file systems.

2. Freeman, Donald E., and Olney R. Perry, *I/O DESIGN: Data Management in Operating Systems*. Rochelle Park, N.J.: Hayden Book Company, Inc., 1977, Chap. 1. This text gives an excellent introduction to file systems. In addition, the authors provide a good perspective as to how file systems fit in the overall scheme of things within the operating environment of a computer system.
3. Hanson, Owen, *Design of Computer Data Files*. Rockville, Md.: Computer Science Press, Inc., 1982, Chap. 1. Hanson's introduction to file systems includes not only definitions, but also an overview of some of the major access method components within a file system.
4. Martin, James, *Computer Data-Base Organization*. 2nd ed. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1975, Chaps. 17, 18. The major feature of these chapters is Martin's ability to discuss many of the trade-offs that need to be considered when designing file systems.

EXERCISES

1. Why are *file attributes* critical to the correct processing of a file?
2. What is the major constraint on overall system performance and throughput? Explain.
3. File systems must make trade-offs between functionality, performance, and resources. Assume that you were designing a file system for a microcomputer. How would these trade-offs affect the methodology and decision making in your design?
4. Assume that an average disk I/O operation takes 20 milliseconds. If a system has three disks, what is the maximum number of disk I/O operations that can be processed in 1 second?
5. Why do most users consider functionality more important that performance?
6. If the file activity ratio is low and the file volatility ratio is high, what kinds of record operations should we try to optimize?
7. How does the *blocking factor* file attribute affect performance? Explain.
8. What is the difference between a field and a record? Explain.
9. Why is the *name* of a file not good enough to locate a particular file? How does the file's *pathname* solve these problems?
10. What are the advantages and disadvantages of having *spanned* records in a file? Explain.

PROGRAMMING EXERCISE

1. Examine the I/O statements in COBOL and one other language. Then define what functions and data structures would be required in a file system to support these languages.